An Architecture for an R Plugin for Gnumeric

Duncan Temple Lang

June 28, 2002

Abstract

We discuss the basic architecture of, and some of the issues related to embedding R within Gnumeric the spreadsheet. We start with describing the elements we might inherit from the Python plugin.

1 Why Gnumeric?

The spreadsheet interface are convenient for certain tasks, and for certain types of users. They allow one to easily layout different elements of analysis. One can print spreadsheets to Postscript and PDF. It is extensible, allowing plugins for different file formats, embedded languages, etc. It has CORBA support.

2 Startup

When the plugin is started, we read the startup file of R code from the package's library.

Additionally, we read the profile not from `R_PROFILE`, but `R_GNUMERIC_PROFILE`. This defaults to `HOME/.gnumeric/rc.R`. Use `R_LoadProfile()`.

3 Defining New Functions

We can do this from within R or from Gnumeric. Each function is specified by

- name,
- category,
- parameter types,
- parameter names,
- help string/documentation,
- the R function (by value or by name?).

Each R function should take a *.gnumeric* argument which allows it to make callbacks to the worksheet from which it was called and also access other functions within the gnumeric application itself.

The argument types are (taken from the gnumeric-python document)

f	A floating point value.
s	A string.
b	A boolean.
r	A Range, e. g. A1:A5 - See 'A'
a	An Array e. g. 1,2,3;4,5,6 (a 3x2 array) - See 'A'
A	Either an Array or a Range.
?	Any type.
|	This designates that the arguments in the token string following this character are optional.

We wil also want dates, etc. And we will approach this using the general, extensible converter mechanism used in the other packages. Additionally, we will probably put a different, more S-like interface such as

```
defineGnumericFunction(name, category, c("integer", "float",
                                         "floatArray", "range"))
```

One calls the functions by the name they were given at declaration time.

Named arguments and optional number of arguments (...) are always a challenge and require some special mechanism.

To call a gnumeric function from R, we use the *.gnumeric* argument. We treat it like an classically object-oriented instance and invoke methods within it.

We want to be able to return the results to R and also to add them to the worksheet.

We may be able to store the `USER_OBJECT_` that is the function in the `FunctionDef` using the `function_def_set_user_dat` This would avoid the use of the static `funclist` and the potentially expensive lookup.

Calls to functions with a fixed number of arguments that supply an incorrect number of arguments in the call are caught by the gnumeric engine.

Define an R "call" function which takes the name of the R function to invoke.

Pass all function calls in the marshalling to an intermediate handler function that actually does the dispatching. This can then determine if the function has a .sheet and/or .cell argument. We can also do this via C code. See the function *gnumeric.hasSheetArgument()*.

Look at the IDL for Gnumeric to get the interface.

A different approach to cells is to return references to cell objects. Then one can use operator overloading to get its value as in

```
sheet[1,2]$value
```

Also, one can then set attributes of the cell in the natural assignment fashion.

```
sheet[1,2]$value <- 10
sheet[1,2]$foreground <- c(65535,0,0)
sheet[1,2]$italic <- TRUE
```

May be worth experimenting with this interface.

4 The Sheet Object

The sheet object can be passed as an argument in all calls. This is an S object of class *GnumericSheet*. One can invoke methods on this object. For example, one can ask it for the dimensions of the sheet's extent. One can also ask it for particular cell values or ranges of these values.

5 Event Loop

As with all embedding, integrating the even loop is an important concern. The standard graphics devices do not get updated on a resize (or when the window is raised to the foreground and X server backing store is not on.)

6 Errors

Need to add an implementation of `jump_now()`.

7 XML

We can generate output in XML format to appear in a Gnumeric worksheet.

8 Installing the Plugin

To make the plugin accessible from gnumeric, you will need to put the `plugin.xml` and `RGnumericPlugin.so` into a directory in the `plugins/` directory that Gnumeric searches. Currently, I don't see a way for us to automatically determine this. Instead, you will have to look for the directory. On my machine, compiling gnumeric from source uses the directory `/usr/local/lib/gnumeric/0.64/plugins/` for this purpose. There is a directory named `applix/` within this. Let's denote this directory by the variable `GNUMERIC_PLUGIN_DIR`. Then, the following

The installation is quite simple. As usual, you will need to arrange to have the application be able to find `libR.so`. You can do this by setting the `LD_LIBRARY_PATH` to point to `R_HOME`/bin. Alternatively, you can add that directory to those that the system loader (ld-so) searches. Within Linux, we can do this by editing the `ld.so.conf` and appending this directory the list. Don't forget to rerun `ldconfig` to rebuild the cache.

9 Questions

How does the call from gnumeric pass control to the interpreter? In other words, which routine in the plugin is invoked? See the file `src/func.h` in the gnumeric source. This file provides numerous routines.

The two routines we want are `function_add_args()` and `function_add_nodes()` which handle the different types of functions. These are distinguished by having fixed parameters types and an arbitrary number arbitrary typed parameters.

When the function is called, it is done by a call to a routine that we pass to the registration function. This routine should have a signature

```
Value * ()(FunctionEvalInfo *ei, Value **args);
```

The `FunctionEvalInfo` contains information about the context in which the function has been invoked and also the `FunctionDefinition`.

How do we signal an error?

10 References

We can store a reference to an R object in a cell and thus hold it around for use in future computations.

11 Examples

To start developing the plugin and to test it, we need some examples. These examples should test the different styles of functions and the different parameter types.

11.1 Random Number Generation

We start with returning a single random value.

We will allow the user to sample a single observation from a Normal and and a bernoulli random variable using the S functions *rnorm()* and *rbinom()*.

```
gnumeric.registerFunction("rbernoulli", "f" , "probability",
                          function(p){rbinom(1,1, p)},
                          "return a value sampled from a Bernoulli random variable.")
```

Since this is called with a fixed number of arguments of specific types (a single real number), we use the invocation mechanism `RGnumeric_fixedArgCall()`. This converts the arguments to R objects and creates a function call. It evaluates this call and then converts the result to a Gnumeric type.

Next, we extend these to return an array of values and to take a range of values as inputs.

11.2

Defining a new function means that it is re-defined each time we recompute the spreadsheet. This makes for interesting synchronization problems and circularity. Thus it should be avoided. We would like to be able to declare a cell as "evaluate once" only.

11.3 Classification

Classification and clustering is an important tool in data analysis. With the integration of R, we can use quite sophisticated models to perform the classification and then use the results to predict new values.

We start with a simple scenario. Somebody first fits a model to a training set of variables and observations. We call this model *myFit()* and assume it is stored (in binary XDR format) in a file named `myFit`. We can load this into the R session using the Gnumeric extension function load(). Then, this fit is available to the R session. We can then define a function to make predictions from this model.

We provide the new observations in columns within a spreadsheet. We define the output column that contains the predictions from these records. Suppose, the predictors are in columns A, B, ..., K. Then, we define a column L which is the result of predicting from the other columns. Suppose we have function named *myPredict()* which has access to the *myFit()* object. Then, the first cell in column L is defined as

```
=myPredict(A1:K1)
```

and then we use the AutoFill facility to copy this cell to other rows.

The function *myPredict()* is defined to accept a range of cells. It then fetches these values and converts them to the appropriate types and performs the prediction.

Regression is a simple case that can be handled with a formula in Gnumeric. Given the estimated coefficients of the linear model, we use these in a regular linear model formula to make the predictions

$$y = \beta_0 + \beta_1 X_1 + \ldots + \beta_k X_k$$

Can we have R write a formula into one or more cells from such a linear model? What about ***`sheet_cell_set_expr()`****?*

Classification and regression trees are more problematic. These are non-parameteric fits and cannot be easily represented by simple formulae. Thus, we want to use R's existing functionality not only to fit these, but also to make predictions from these.

12 Statistical Functionality

Gnumeric already has support for some of the standard statistical methodologies (z- and t-tests, regression, sampling, etc.) These have graphical interfaces. It would be interesting to see if we can provide interfaces for other statistical functionality and to do this from within R.

Also, we should consider ways to output R objects in "display" format within the Gnumeric spreadsheet cells.

13 Graphics Devices

Given the changes to the X11 device t support graphics devices as "inlined" plugins within a Netscape page/document, we should be able to use a cell in the gnumeric spreadsheet as a container/canvas for an R graphics device. In this way, we can embed R plots within the spreadsheet interface and dynamically update them as part of the recalculation cycle. This would allow the rich graphics model and the wide variety of plots that R is capable of producing be directly and simply imported into gnumeric. This has potentially a wide audience as it would reduce the dependency by gnumeric on Guppi, etc.

14 Gnumeric in R

The other side of this interface is to run gnumeric as a plugin to R. This would allow a user to start Gnumeric as a data editor or front-end for R at any point in the R session. This should be reasonably simple to arrange, by having gnumeric as a shared library.

15 Remote Interface: CORBA

We have CORBA facilities in R and Gnumeric offers a CORBA interface. The issues we have to sort out are

- locating the gnumeric object when not using the gnome desktop and its naming services; and
- using ORBit within R.

These are orthogonal and non-dependent issues.

16 Gtk Bindings for R

I am moving closer to starting to implement bindings from R to Gtk. This would allow us to programmatically from within R create new tools that utilized the growing number of Gtk widgets. For example, we would be able to build dialogs, etc. for ggobi. Additionally, we might create a new interface for ggobi that included one or more R graphics devices and ggobi elements/components.

It would be nice to have a (partially) automated mechanism to implement the interface. I have been considering developing a version of a SWIG-like tool perhaps using lcc. I have already modified lcc to identify and potentially remove non-local variables for the threading setup. A SWIG-like tool would not be too hard to develop with this, but does require thinking about the pass-by-value and pass-by-reference semantics that may be undetectable from C.